Parallel & Distributed Computing

UCS645

Assignment - 2

Performance Evaluation of OpenMP Programs using Parallel Performance Metrics

Name - Kartik Goel
Roll No. - 102303182

System Environment: Linux (Ubuntu), GCC Compiler with OpenMP Support

AIM:

• Implement basic OpenMP parallel programs.

• Measure execution time using omp_get_wtime().

• Compute speedup, efficiency, and cost metrics.

• Understand strong vs weak scaling using Amdahl's and Gustafson's laws.

• Identify performance bottlenecks such as load imbalance, synchronization overhead, false sharing, and memory bandwidth saturation.

• Gain introductory exposure to performance profiling tools.

Question 1 - Molecular Dynamics - Force Calculation

Using code:-

```
auto compute_potential_and_force_parallel(Particles& particles, double& total_energy, int num_threads) -> std::chrono::duration<double, std::milli> {
        total_energy = 0.0;
        int n{static_cast<int>(particles.position.size())};

        #pragma omp parallel for
        for(int i = 0; i<n; ++i){
                particles.force[i] = {0.0, 0.0, 0.0};
        }

        total_energy = 0.0;
        std::cout << std::format("With {} threads\n", num_threads);
        auto th_start{std::chrono::steady_clock::now()};
        #pragma omp parallel for reduction(+:total_energy) schedule(dynamic) num_threads(num_threads)
        for(int i = 0; i<n;++i) {
                vec3_t current_force{0.0, 0.0, 0.0};
                for(int j = 0; j<n; ++j) {
                        if(i == j) continue;
                        vec3_t delta{particles.position[i] - particles.position[j]};

                        double r2{SQUARE(delta.x) + SQUARE(delta.y) + SQUARE(delta.z)};

                        if (r2 < 1e-10) continue;
                        double r2_inv{1.0/r2};
                        double s2_inv{SIGMASQ*r2_inv};
                        double s6_inv{SQUARE(s2_inv) * s2_inv};
                        double s12_inv{SQUARE(s6_inv)};

                        double pair_energy{4 * EPSILON * (s12_inv - s6_inv)};
                        total_energy += pair_energy;

                        double force_scalar{(24.0 * EPSILON * r2_inv) * (2.0 * s12_inv - s6_inv)};
                        vec3_t force_vec{delta * force_scalar};

                        current_force += force_vec;
                }
                particles.force[i] = current_force;
        }
        auto th_end{std::chrono::steady_clock::now()};
        std::chrono::duration<double, std::milli> ms{th_end - th_start};
        std::cout << std::format("Execution time: {:.2f}ms\n", ms.count());
        std::cout << std::format("Total Energy: {}\n", total_energy * 0.5);
        return ms;
}
```

ON EXECUTION:-

Number of Threads	Execution Time (ms)	Total Energy	Speed Up	Throughput	Efficiency
1	246.52	1.184670917705871e+23	1.00x	NA	NA
2	121.46	1.1846709177059208e+23	2.03x	82.33	1.01
4	60.68	1.1846709177059488e+23	4.06x	164.81	1.02
6	42.79	1.1846709177059576e+23	5.76x	233.71	0.96
8	35.48	1.1846709177059638e+23	6.95x	281.86	0.87
10	29.02	1.1846709177059654e+23	8.50x	344.62	0.85
12	25.81	1.184670917705967e+23	9.55x	387.39	0.80

14	25.19	1.1846709177059695e+23	9.79x	397.01	0.70
16	24.06	1.184670917705972e+23	10.24x	415.56	0.64
18	23.77	1.18467091770597e+23	10.37x	420.68	0.58
20	24.69	1.1846709177059728e+23	9.98x	404.96	0.50
22	24.39	1.1846709177059723e+23	10.11x	410.03	0.46
24	24.84	1.1846709177059733e+23	9.92x	402.50	0.41

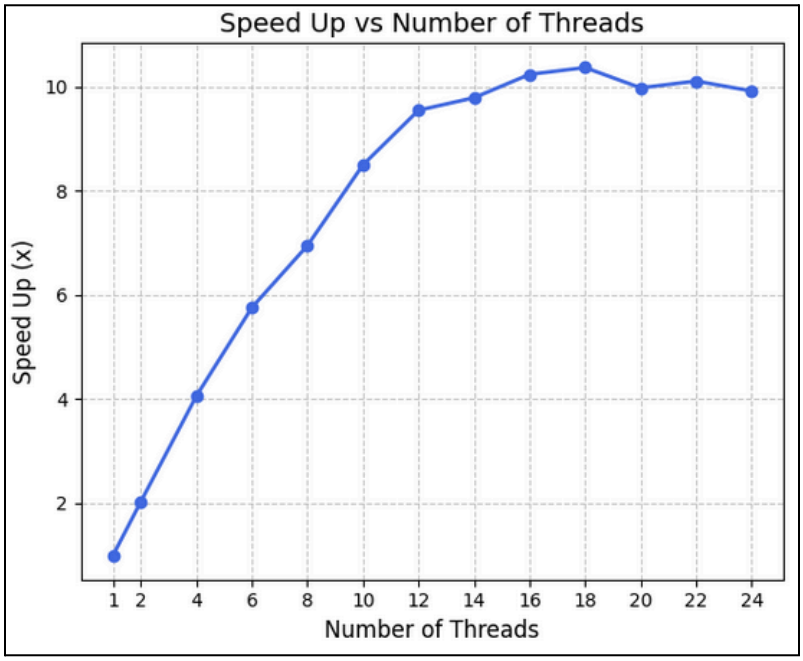
INFERENCE:-

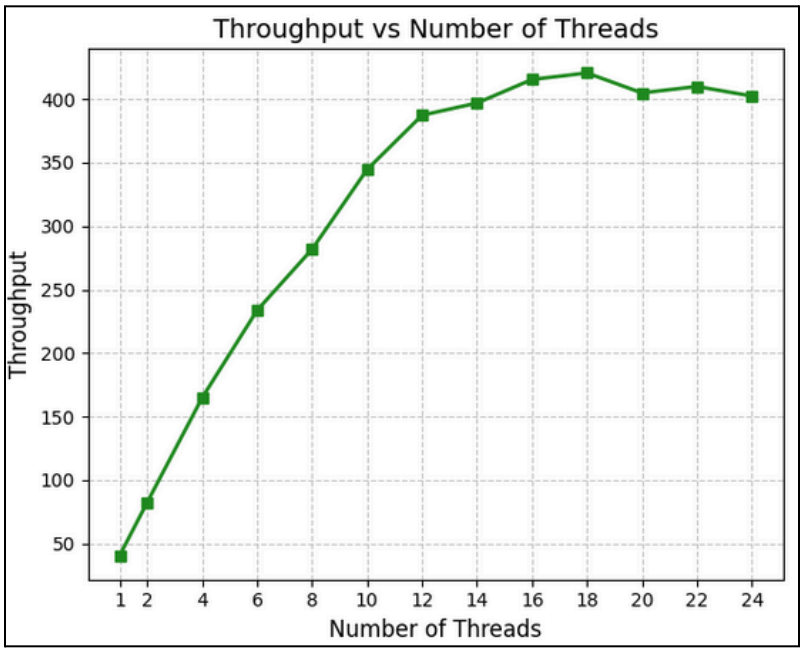
The OpenMP implementation successfully parallelizes the O(N2) force calculations and correctly handles race conditions via reduction, as evidenced by the consistently invariant total energy (1.18e+23) across all test runs. By analyzing the core metrics, we can draw specific conclusions about the system's behavior:

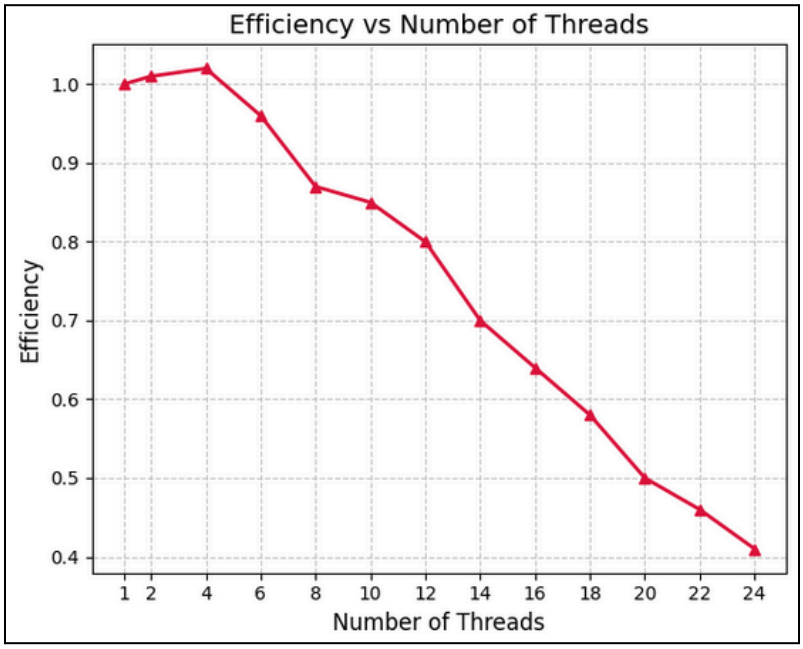
1. **Speedup (The Acceleration):** The program exhibits near-ideal, linear scaling up to 4 threads, achieving a 4.06x speedup. Beyond this point, the scaling becomes sub-linear, eventually hitting a hard performance ceiling at 18 threads with a maximum speedup of 10.37x. Pushing the thread count higher (20–24 threads) causes the speedup to actively regress, proving that the time spent managing threads now outweighs the computational benefits.

2. **Throughput (The Processing Capacity):** Mirroring speedup, the system throughput climbs from a single-threaded baseline of 40.56 to a peak of **420.68** at 18 threads. This peak represents the physical saturation point of the hardware. The inability to push throughput higher indicates that shared architectural resources—such as the memory bus streaming the 3D particle coordinates or the CPU's cache bandwidth—are fully saturated.
3. **Efficiency (The Resource Utilization):** This is the most revealing metric for identifying overhead. Efficiency remains near-perfect (1.02) at 4 threads but drops rapidly as the thread count increases, falling to 0.58 at the peak speedup (18 threads) and plummeting to 0.41 by 24 threads. This steep degradation is driven by:
    a. **Lock Contention:** Resolving Newton's third law requires threads to write to shared force arrays. As the thread count grows, atomic locks become highly contested, forcing threads to idle while waiting their turn.
    b. **Context Switching:** Once the thread count exceeds the physical cores available, the operating system is forced to continuously swap thread states in and out of the CPU, wasting cycles that should be spent on potential energy calculations.

While 18 threads yield the absolute fastest execution time, the data suggests the optimal spot is between 8 and 12 threads. In this range, the program delivers substantial speedup (6.95x to 9.55x) while maintaining strong efficiency (0.87 to 0.80), striking the best balance between execution speed and optimal CPU resource allocation.




Throughput vs Number of Threads
Throughput
Number of Threads
50
100
150
200
250
300
350
400
1 2 4 6 8 10 12 14 16 18 20 22 24


Efficiency vs Number of Threads
Efficiency
Number of Threads
0.4
0.5
0.6
0.7
0.8
0.9
1.0
1 2 4 6 8 10 12 14 16 18 20 22 24

OBSERVATION FROM PERF STAT:

```
Performance counter stats for './molecular_dynamics':

         5,196.23 msec task-clock                       #    6.921 CPUs utilized
            1,508      context-switches                 #  290.211 /sec
              526      cpu-migrations                   #  101.227 /sec
              644      page-faults                      #  123.936 /sec
   26,070,304,402      cpu_atom/instructions/           #    1.73  insn per cycle              (28.75%)
   49,667,683,321      cpu_core/instructions/           #    2.48  insn per cycle              (67.09%)
   15,037,882,441      cpu_atom/cycles/                 #    2.894 GHz                         (28.67%)
   20,001,770,288      cpu_core/cycles/                 #    3.849 GHz                         (67.09%)
    2,449,623,310      cpu_atom/branches/               #  471.423 M/sec                       (28.78%)
    4,637,777,528      cpu_core/branches/               #  892.528 M/sec                       (67.09%)
          849,428      cpu_atom/branch-misses/          #    0.03% of all branches             (28.84%)
          677,631      cpu_core/branch-misses/          #    0.01% of all branches             (67.09%)
            TopdownL1 (cpu_core)                 #     41.3 %  tma_backend_bound
                                                 #      1.0 %  tma_bad_speculation
                                                 #      6.7 %  tma_frontend_bound
                                                 #     51.0 %  tma_retiring             (67.09%)
                                                 #      0.5 %  tma_bad_speculation
                                                 #     34.7 %  tma_retiring             (28.84%)
                                                 #     64.4 %  tma_backend_bound
                                                 #      0.4 %  tma_frontend_bound       (28.83%)

      0.750751512 seconds time elapsed

      5.175531000 seconds user
      0.029223000 seconds sys
```

1. Resource Utilization:-

Parallel Efficiency: With 6.921 CPUs utilized, the system is highly parallelized, effectively distributing the 5.19 seconds of task-clock time across nearly 7 processor cores simultaneously during the brief 0.75-second elapsed physical time.

Hybrid Execution: The workload is actively split between Performance-cores (running at a fast 3.849 GHz) and Efficiency/Atom-cores (running at 2.894 GHz).

2. Instruction Efficiency:-

P-Core Performance: The Performance cores achieve an impressive IPC (Instructions Per Cycle) of 2.48, showing highly efficient throughput for this math-heavy task. A very healthy 51.0% of operations are successfully completed without stalling.

E-Core Lag: The Efficiency (Atom) cores are slower, with an IPC of 1.73, meaning they process fewer instructions per clock tick compared to the P-cores.

Branch Prediction: Both core types show near-perfect branch prediction, with branch-misses at a negligible 0.01% (P-cores) and 0.03% (E-cores), meaning the CPU is rarely guessing wrong on your loop conditions.

3. Primary Bottlenecks:-

Backend Bound (41.3% on P-Cores): The frontend is feeding instructions efficiently (only 6.7% frontend bound), but the P-cores are stalling in the backend. This usually means the execution units are waiting for data to arrive from memory (cache misses) to calculate the forces.

Severe Backend Bound (64.4% on Atom): The Efficiency cores are struggling significantly more with the backend. They are spending the vast majority of their time stalled and waiting on memory access or execution resources, resulting in a much lower retiring rate (34.7%).

Question 2: Bioinformatics - DNA Sequence Alignment (Smith-Waterman)

Using Code:-

```
auto dna_sequence_allignment_parallel(const std::string& seqA, const std::string& seqB, int tile_size) -> std::chrono::duration<double, std::milli> {
        int rows{static_cast<int>(seqA.length() + 1)};
        int cols{static_cast<int>(seqB.length() + 1)};

        std::vector<std::vector<int>> matrix(rows, std::vector<int>(cols, 0));

        int n_block_rows{(rows - 1 + tile_size - 1) / tile_size};
        int n_block_cols{(cols - 1 + tile_size - 1) / tile_size};

        std::vector<int> deps(n_block_rows * n_block_cols, 0);

        int dummy_root = 0;

        int global_max_score{0};

        auto start{std::chrono::steady_clock::now()};
        #pragma omp parallel
        {
                #pragma omp single
                {
                        for (int i = 0; i < n_block_rows; ++i) {
                                for (int j = 0; j < n_block_cols; ++j) {

                                        int self_idx = i * n_block_cols + j;
                                        int* self = &deps[self_idx];

                                        int* top = (i == 0) ? &dummy_root : &deps[(i-1) * n_block_cols + j];

                                        int* left = (j == 0) ? &dummy_root : &deps[i * n_block_cols + (j-1)];

                                        #pragma omp task \
                                        depend(in: *top) \
                                        depend(in: *left) \
                                        depend(out: *self)
                                        {
                                                process_tile(i, j, seqA, seqB, tile_size, matrix, global_max_score);
                                        }
                                }
                        }
                }
        }
        auto end{std::chrono::steady_clock::now()};
        std::chrono::duration<double, std::milli> ms{end-start};
        std::cout << std::format("Score: {}\n", global_max_score);
        std::cout << std::format("Execution time: {:.2f}ms\n", ms.count());
        return ms;
}
```

ON EXECUTION:

Number of Threads	Tile Size	Execution Time (ms)	Speed Up	Throughput	Efficiency
1	N/A	186.83	1.00x	N/A	N/A
2	32 x 32	145.06	1.29x	689,365.05	0.64
2	64 x 64	80.12	2.33x	1,248,100.48	1.17
2	96 x 96	71.98	2.60x	1,389,241.45	1.30
2	128 x 128	67.68	2.76x	1,477,517.14	1.38
2	192 x 192	65.23	2.86x	1,532,980.07	1.43
2	256 x 256	60.82	3.07x	1,644,187.31	1.54
2	512 x 512	54.21	3.45x	1,844,714.89	1.72
4	32 x 32	111.62	1.67x	895,882.35	0.42
4	64 x 64	51.20	3.65x	1,953,050.04	0.91

4	96 x 96	42.72	4.37x	2,340,669.95	1.09
4	128 x 128	41.02	4.55x	2,437,813.27	1.14
4	192 x 192	37.89	4.93x	2,639,449.44	1.23
4	256 x 256	36.40	5.13x	2,746,893.76	1.28
4	512 x 512	35.09	5.32x	2,850,048.19	1.33
6	32 x 32	82.70	2.26x	1,209,217.74	0.38
6	64 x 64	39.76	4.70x	2,514,785.49	0.78
6	96 x 96	33.67	5.55x	2,969,656.97	0.92
6	128 x 128	33.54	5.57x	2,981,874.67	0.93
6	192 x 192	30.00	6.23x	3,332,953.49	1.04
6	256 x 256	27.88	6.70x	3,586,484.89	1.12
6	512 x 512	26.72	6.99x	3,741,823.74	1.17

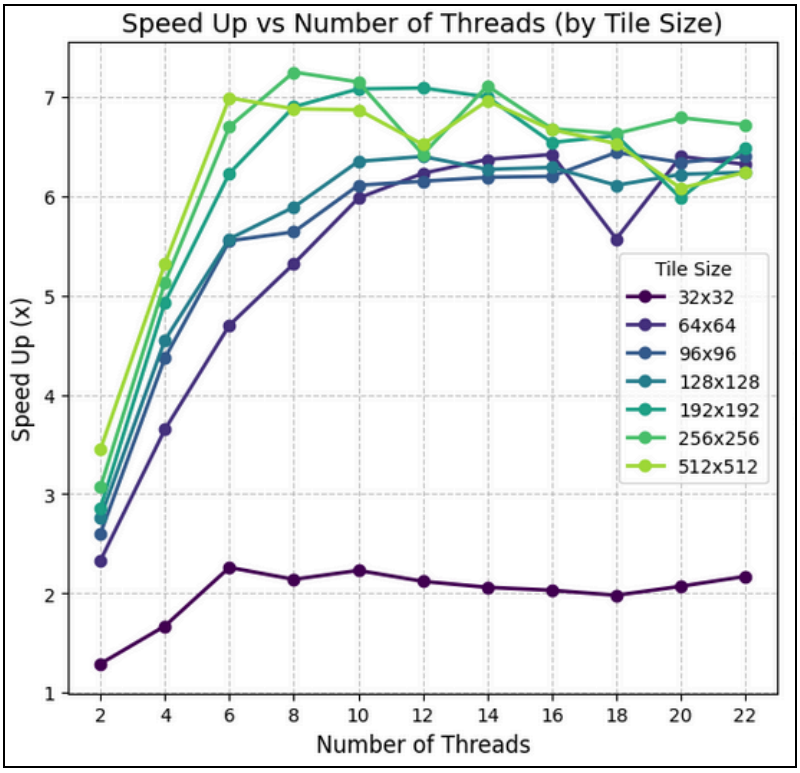
8	32 x 32	87.18	2.14x	1,147,030.39	0.27
8	64 x 64	35.14	5.32x	2,845,517.13	0.66
8	96 x 96	33.10	5.64x	3,021,120.02	0.71
8	128 x 128	31.71	5.89x	3,153,120.41	0.74
8	192 x 192	27.08	6.90x	3,692,675.05	0.86
8	256 x 256	25.78	7.25x	3,879,674.53	0.91
8	512 x 512	27.15	6.88x	3,682,829.43	0.86
10	32 x 32	83.66	2.23x	1,195,305.08	0.22
10	64 x 64	31.26	5.98x	3,198,475.68	0.60
10	96 x 96	30.57	6.11x	3,271,162.92	0.61
10	128 x 128	29.42	6.35x	3,399,239.14	0.64
10	192 x 192	26.40	7.08x	3,788,392.66	0.71

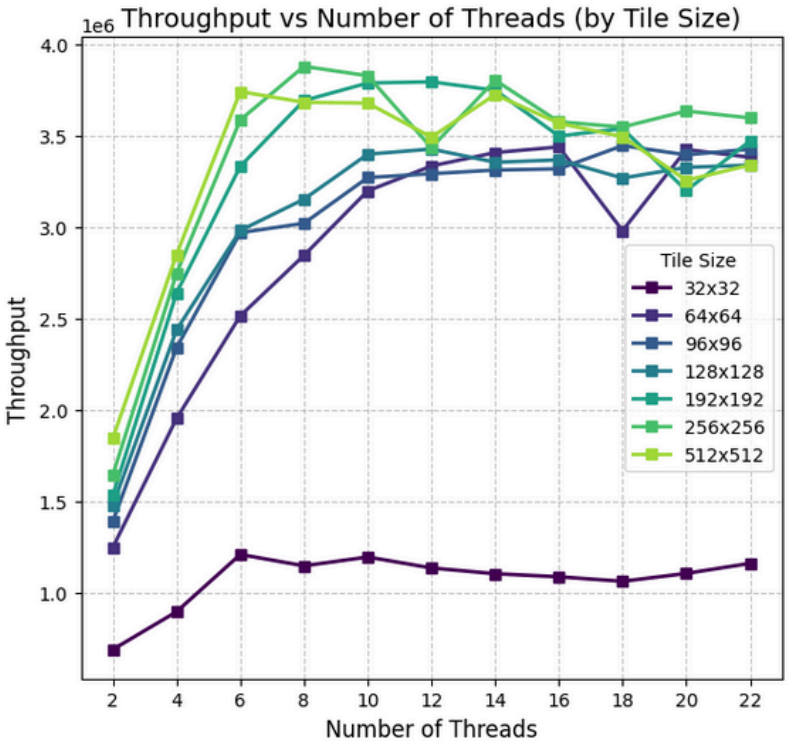
10	256 x 256	26.12	7.15x	3,828,515.58	0.72
10	512 x 512	27.18	6.87x	3,678,800.54	0.69
12	32 x 32	88.08	2.12x	1,135,376.45	0.18
12	64 x 64	29.99	6.23x	3,334,608.04	0.52
12	96 x 96	30.37	6.15x	3,292,564.25	0.51
12	128 x 128	29.18	6.40x	3,427,186.38	0.53
12	192 x 192	26.35	7.09x	3,795,203.82	0.59
12	256 x 256	29.11	6.42x	3,434,688.83	0.53
12	512 x 512	28.64	6.52x	3,492,218.81	0.54
14	32 x 32	90.59	2.06x	1,103,853.00	0.15
14	64 x 64	29.34	6.37x	3,408,009.64	0.45
14	96 x 96	30.19	6.19x	3,312,301.21	0.44

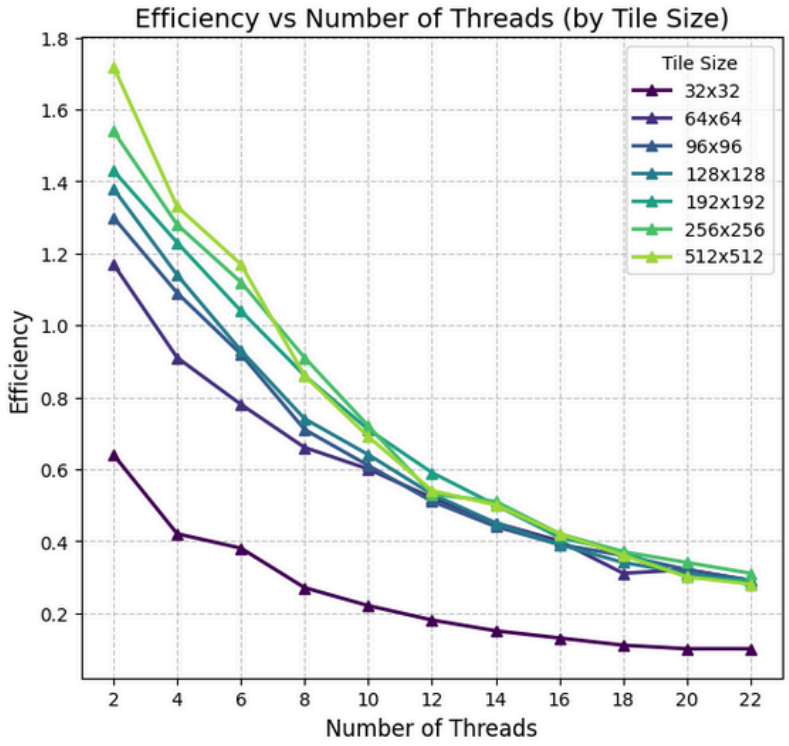
14	128 x 128	29.81	6.27x	3,355,029.64	0.45
14	192 x 192	26.67	7.00x	3,748,999.53	0.50
14	256 x 256	26.29	7.11x	3,804,229.19	0.51
14	512 x 512	26.84	6.96x	3,725,251.94	0.50
16	32 x 32	91.98	2.03x	1,087,200.04	0.13
16	64 x 64	29.08	6.42x	3,438,996.62	0.40
16	96 x 96	30.13	6.20x	3,319,193.46	0.39
16	128 x 128	29.69	6.29x	3,367,738.37	0.39
16	192 x 192	28.58	6.54x	3,498,581.36	0.41
16	256 x 256	27.96	6.68x	3,576,380.71	0.42
16	512 x 512	28.02	6.67x	3,568,775.31	0.42
18	32 x 32	94.14	1.98x	1,062,260.64	0.11

18	64 x 64	33.55	5.57x	2,981,056.52	0.31
18	96 x 96	29.02	6.44x	3,445,903.18	0.36
18	128 x 128	30.59	6.11x	3,268,554.83	0.34
18	192 x 192	28.25	6.61x	3,539,255.10	0.37
18	256 x 256	28.19	6.63x	3,547,793.80	0.37
18	512 x 512	28.61	6.53x	3,494,695.05	0.36
20	32 x 32	90.46	2.07x	1,105,505.49	0.10
20	64 x 64	29.20	6.40x	3,424,592.91	0.32
20	96 x 96	29.45	6.34x	3,395,592.31	0.32
20	128 x 128	30.06	6.22x	3,327,056.18	0.31
20	192 x 192	31.25	5.98x	3,200,042.91	0.30
20	256 x 256	27.51	6.79x	3,635,446.58	0.34

20	512 x 512	30.71	6.08x	3,256,402.03	0.30
22	32 x 32	86.17	2.17x	1,160,430.06	0.10
22	64 x 64	29.56	6.32x	3,382,404.47	0.29
22	96 x 96	29.19	6.40x	3,426,289.25	0.29
22	128 x 128	29.95	6.24x	3,339,342.26	0.28
22	192 x 192	28.81	6.49x	3,471,527.31	0.29
22	256 x 256	27.80	6.72x	3,597,325.98	0.31
22	512 x 512	29.94	6.24x	3,340,158.73	0.28


Speed Up vs Number of Threads (by Tile Size)
Speed Up (x)
Number of Threads
Tile Size
32x32
64x64
96x96
128x128
192x192
256x256
512x512


Throughput vs Number of Threads (by Tile Size)
1e6
Throughput
Number of Threads
Tile Size
32x32
64x64
96x96
128x128
192x192
256x256
512x512



INFERENCE:-

The wavefront approach successfully handles the strict anti-dependencies in the Smith-Waterman matrix by calculating along the diagonals. A review of the results reveals a few key takeaways about how the system handles the workload:

1. **Tile Size is Critical** The size of the data blocks (tiles) drastically dictates performance by balancing thread synchronization with cache memory.
    a. **Too Small (32x32):** These perform poorly, peaking at only a 2.26x speedup. The threads finish the math so quickly that they spend most of their time idling at the OpenMP barriers, waiting for the next diagonal to start.
    b. **Optimal Balance (256x256):** This size performs the best, hitting the fastest execution time (25.78ms) and peak speedup (7.25x) at 8 threads. It gives the CPU just enough work to minimize barrier wait times without spilling out of the fast L1/L2 cache.
    c. **Too Large (512x512):** Pushing the tile size higher causes a slight drop in performance, likely because the blocks exceed the local cache capacity and force the system to rely on slower memory.
2. **Scaling Limits** The setup scales well up to 8–10 threads, but adding more actively hurts efficiency, which drops to 0.31 by 22 threads. Since wavefront

processing requires all threads to finish their current diagonal before moving on, the fast Performance (P) cores end up sitting idle at the synchronization barriers, waiting for the slower Efficiency (E) cores to finish their work.

3. **Hardware Bottlenecks** The CPU telemetry shows exactly where the processing pipeline gets stuck:
    a. **Memory Stalls:** The dynamic programming algorithm constantly needs to fetch neighboring penalty values. As a result, the P-cores are stalled 41.3% of the time (backend bound), and the E-cores stall 64.4% of the time, just waiting for data to arrive from memory.
    b. **Smooth Logic:** The branch prediction is near-perfect (only 0.01% misses on P-cores), meaning the CPU isn't struggling with the math or logic—it is just starved for data.

OBSERVATION FROM PERF STAT:

```
Performance counter stats for './dna_sequence_allignment':

        29,045.33 msec task-clock                       #    4.323 CPUs utilized
           438,554      context-switches                 #   15.099 K/sec
             5,104      cpu-migrations                   #  175.725 /sec
           126,425      page-faults                      #    4.353 K/sec
   138,980,458,282      cpu_atom/instructions/           #    1.53  insn per cycle              (21.63%)
   200,685,948,718      cpu_core/instructions/           #    1.70  insn per cycle              (75.30%)
    90,761,935,672      cpu_atom/cycles/                 #    3.125 GHz                         (21.59%)
   118,024,701,357      cpu_core/cycles/                 #    4.063 GHz                         (75.30%)
    14,578,079,276      cpu_atom/branches/               #  501.908 M/sec                       (21.61%)
    21,147,789,139      cpu_core/branches/               #  728.096 M/sec                       (75.30%)
       344,394,121      cpu_atom/branch-misses/          #    2.36% of all branches             (21.59%)
       419,443,523      cpu_core/branch-misses/          #    1.98% of all branches             (75.30%)
            TopdownL1 (cpu_core)                 #     54.2 %  tma_backend_bound
                                                 #      7.3 %  tma_bad_speculation
                                                 #      7.8 %  tma_frontend_bound
                                                 #     30.6 %  tma_retiring             (75.30%)
                                                 #     25.9 %  tma_bad_speculation
                                                 #     31.5 %  tma_retiring             (21.61%)
                                                 #     34.5 %  tma_backend_bound
                                                 #      8.0 %  tma_frontend_bound       (21.59%)

       6.718282760 seconds time elapsed

      26.026832000 seconds user
       3.198298000 seconds sys
```

1. Resource Utilization:-

Parallel Efficiency: With 4.323 CPUs utilized, the system is moderately parallelized, distributing the 29.04 seconds of task-clock time across just over 4 processor cores during the 6.72-second elapsed physical time.

Hybrid Execution: The workload is actively split between Performance-cores (running at a very fast 4.063 GHz) and Efficiency/Atom-cores (running at 3.125 GHz).

2. Instruction Efficiency:-

P-Core Performance: The Performance cores achieve a moderate IPC (Instructions Per Cycle) of 1.70, showing lower throughput compared to simpler mathematical loops. Only 30.6% of operations are successfully retiring (completing without stalling).

E-Core Lag: The Efficiency (Atom) cores are slightly slower, with an IPC of 1.53, meaning they process fewer instructions per clock tick compared to the P-cores, with a similar retiring rate of 31.5%.

Branch Prediction: Both core types struggle noticeably with branch prediction. Branch misses sit at 1.98% for P-cores and 2.36% for E-cores. Because algorithms like Smith-Waterman rely heavily on conditional branching (e.g., continually checking which of four neighboring penalties is the maximum), the CPU is frequently guessing the wrong path.

3. Primary Bottlenecks:-

Severe Backend Bound (54.2% on P-Cores): The Performance cores are heavily stalling in the backend. With over half of the execution time choked here, the processing units are severely starved for data, spending most of their time waiting on memory access to fetch neighboring matrix values for the dynamic programming grid.

Dual Bottleneck & Bad Speculation (on Atom): The Efficiency cores are facing a compounded issue. Not only are they Backend Bound (34.5%) while waiting for memory, but they also suffer from a massive Bad Speculation penalty (25.9%). This means over a quarter of the E-cores' pipeline time is entirely wasted executing instructions for the wrong branch path, only to throw that work away and start over.

Question 3: Scientific Computing - Heat Diffusion Simulation

Using Code:

```
auto run_simulation(int num_threads, omp_sched_t sched_type, int chunk_size) -> SimResult {
        std::vector<double> T(N * N, 0.0);
        std::vector<double> T_next(N * N, 0.0);
        int center{N / 2};
        int radius{N / 10};
        #pragma omp parallel for schedule(static) num_threads(num_threads)
        for (int i = 0; i < N; ++i) {
                for (int j = 0; j < N; ++j) {
                        if ((i - center)*(i - center) + (j - center)*(j - center) < radius*radius) {
                                T[i * N + j] = 100.0;
                        } else {
                                T[i * N + j] = 0.0;
                        }
                }
        }
        double cx {ALPHA * DT / (DX * DX)};
        double cy {ALPHA * DT / (DX * DX)};

        omp_set_num_threads(num_threads);
        omp_set_schedule(sched_type, chunk_size);

        auto start_time{std::chrono::high_resolution_clock::now()};

        for (int step{0}; step < STEPS; ++step) {

        #pragma omp parallel for schedule(runtime)
                for (int i = 1; i < N - 1; ++i) {
                        for (int j = 1; j < N - 1; ++j) {
                                int idx = i * N + j;
                                int up = (i - 1) * N + j;
                                int down = (i + 1) * N + j;
                                int left = i * N + (j - 1);
                                int right = i * N + (j + 1);

                                T_next[idx] = T[idx] +
                                        cx * (T[up] - 2 * T[idx] + T[down]) +
                                        cy * (T[left] - 2 * T[idx] + T[right]);
                        }
                }
                std::swap(T, T_next);
        }

        auto end_time{std::chrono::high_resolution_clock::now()};
        std::chrono::duration<double, std::milli> duration{end_time - start_time};

        double total_energy{0.0};
        #pragma omp parallel for reduction(+:total_energy) num_threads(num_threads)
        for (int i = 0; i < N * N; ++i) {
                total_energy += T[i];
        }

        return { duration.count(), total_energy };
}
```

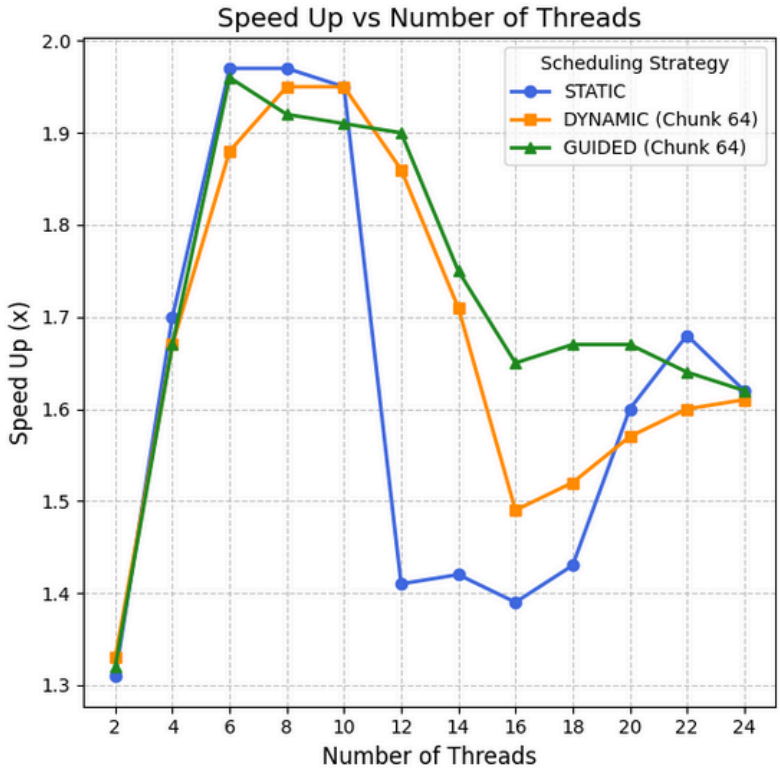
ON EXECUTION:

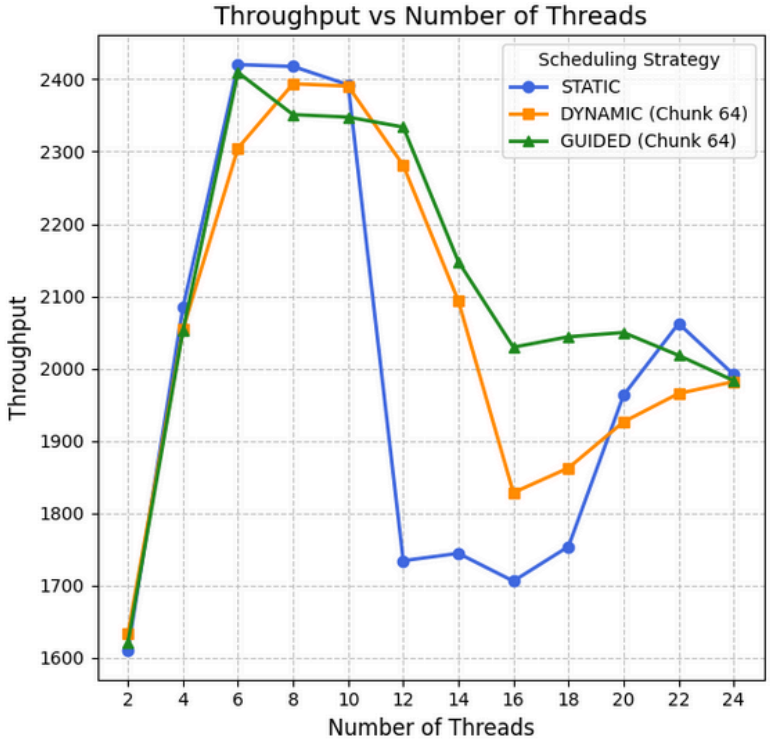
Number of Threads	Scheduling Strategy	Execution Time (ms)	Speed Up	Throughput	Efficiency
1	N/A	1629.57	1.00x	NA	NA
2	STATIC	1241.89	1.31x	1610.45	0.66
2	DYNAMIC (Chunk 64)	1224.86	1.33x	1632.85	0.67
2	GUIDED (Chunk 64)	1234.18	1.32x	1620.51	0.66
4	STATIC	959.48	1.70x	2084.46	0.42
4	DYNAMIC (Chunk 64)	973.58	1.67x	2054.27	0.42
4	GUIDED (Chunk 64)	974.55	1.67x	2052.23	0.42
6	STATIC	826.36	1.97x	2420.27	0.33

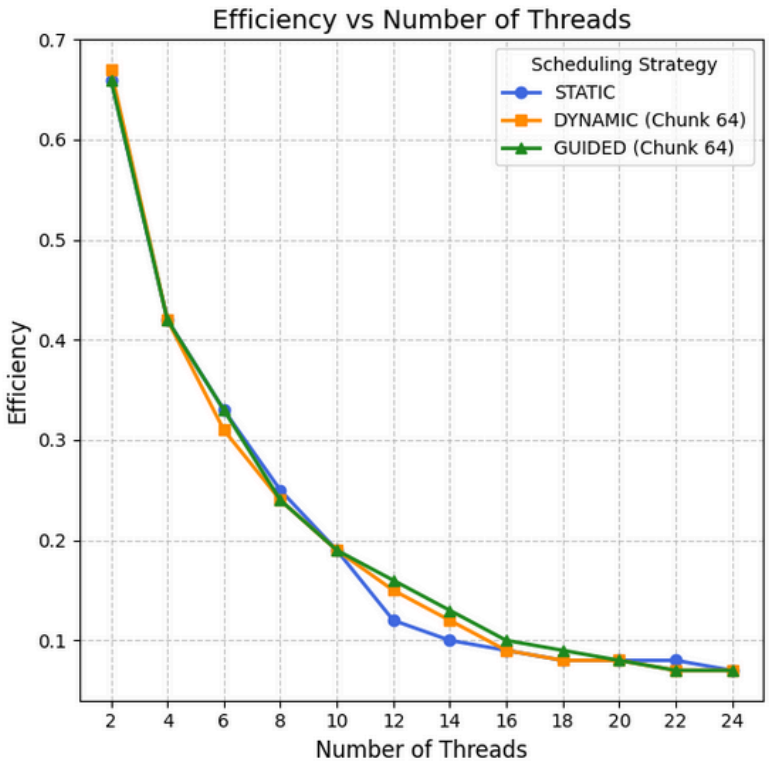
6	DYNAMIC (Chunk 64)	867.70	1.88x	2304.95	0.31
6	GUIDED (Chunk 64)	830.08	1.96x	2409.42	0.33
8	STATIC	827.22	1.97x	2417.73	0.25
8	DYNAMIC (Chunk 64)	835.50	1.95x	2393.79	0.24
8	GUIDED (Chunk 64)	850.66	1.92x	2351.11	0.24
10	STATIC	835.99	1.95x	2392.37	0.19
10	DYNAMIC (Chunk 64)	836.65	1.95x	2390.48	0.19
10	GUIDED (Chunk 64)	851.91	1.91x	2347.67	0.19
12	STATIC	1153.20	1.41x	1734.30	0.12
12	DYNAMIC (Chunk 64)	876.83	1.86x	2280.95	0.15

12	GUIDED (Chunk 64)	856.94	1.90x	2333.87	0.16
14	STATIC	1146.49	1.42x	1744.46	0.10
14	DYNAMIC (Chunk 64)	955.21	1.71x	2093.78	0.12
14	GUIDED (Chunk 64)	931.09	1.75x	2148.02	0.13
16	STATIC	1172.26	1.39x	1706.10	0.09
16	DYNAMIC (Chunk 64)	1093.86	1.49x	1828.39	0.09
16	GUIDED (Chunk 64)	985.47	1.65x	2029.49	0.10
18	STATIC	1140.33	1.43x	1753.88	0.08
18	DYNAMIC (Chunk 64)	1073.79	1.52x	1862.56	0.08
18	GUIDED (Chunk 64)	978.49	1.67x	2043.96	0.09

20	STATIC	1018.80	1.60x	1963.10	0.08
20	DYNAMIC (Chunk 64)	1038.28	1.57x	1926.27	0.08
20	GUIDED (Chunk 64)	975.70	1.67x	2049.82	0.08
22	STATIC	969.80	1.68x	2062.28	0.08
22	DYNAMIC (Chunk 64)	1017.56	1.60x	1965.48	0.07
22	GUIDED (Chunk 64)	990.98	1.64x	2018.21	0.07
24	STATIC	1004.17	1.62x	1991.70	0.07
24	DYNAMIC (Chunk 64)	1009.16	1.61x	1981.85	0.07
24	GUIDED (Chunk 64)	1008.40	1.62x	1983.33	0.07


Speed Up vs Number of Threads
Scheduling Strategy
STATIC
DYNAMIC (Chunk 64)
GUIDED (Chunk 64)
Speed Up (x)
Number of Threads
2.0
1.9
1.8
1.7
1.6
1.5
1.4
1.3
2
4
6
8
10
12
14
16
18
20
22
24


Throughput vs Number of Threads
Scheduling Strategy
STATIC
DYNAMIC (Chunk 64)
GUIDED (Chunk 64)
Throughput
Number of Threads
2400
2300
2200
2100
2000
1900
1800
1700
1600
2
4
6
8
10
12
14
16
18
20
22
24



INFERENCE:

Based on the execution data and the generated graphs for the finite difference method, we can see a very different performance profile compared to previous tasks. The most striking takeaway is how the choice of OpenMP scheduling strategy entirely dictates how the program survives higher thread counts on a hybrid CPU.

1. **The Overall Performance Ceiling (Memory Bottleneck)** Unlike the massive 10x speedups seen in purely computational tasks, this heat diffusion simulation hits a hard brick wall. Across all strategies, the maximum speedup barely touches **1.97x** (around 6 to 10 threads). This indicates the algorithm is severely memory-bound. Calculating the next temperature of a grid point requires fetching multiple neighboring points from memory. The CPU's arithmetic units are likely sitting idle, starved for data, because the system's memory bandwidth is fully saturated almost immediately.

2. **The 12-Thread Collapse (The Problem with STATIC)** If you look at the blue line (STATIC scheduling) on the Speed Up and Throughput graphs, there is a massive collapse right at 12 threads. The execution time jumps from 835ms to 1153ms.
    a. **Why it happens:** Static scheduling blindly chops the 2D grid into equal, fixed-size chunks and hands them to the threads. However, your CPU has

a mix of fast Performance (P) cores and slower Efficiency (E) cores. By 12 threads, the system is forced to heavily utilize those slower E-cores. The fast P-cores finish their chunks quickly and then sit entirely idle at the synchronization barrier, waiting for the slower E-cores to finish their massive, equally-sized chunks.

3. **The Adaptive Advantage (DYNAMIC & GUIDED)** The orange (DYNAMIC) and green (GUIDED) lines clearly demonstrate how to solve this load imbalance.
    a. **Dynamic (Chunk 64):** Instead of one massive chunk, it breaks the work into smaller blocks of 64. When a fast P-core finishes a block, it immediately grabs the next one from the queue, naturally doing more work than the slower E-cores.
    b. **Guided (Chunk 64):** This performs the best under heavy thread loads (14–18 threads). It starts by handing out large chunks to reduce overhead, and progressively shrinks the chunk size as the queue empties. This perfectly balances the workload across the P-cores and E-cores, completely avoiding the catastrophic 12-thread drop-off seen in the static approach.
4. **Plunging Efficiency** Because the memory bus is completely saturated early on, adding more threads simply creates overhead without adding processing power. By 24 threads, parallel efficiency collapses to a dismal **0.07**. The system is spending massive amounts of energy (and time managing the threads) for absolutely no performance gain.

OBSERVATION FROM PERF STAT:

```
Performance counter stats for './heat_simulation':

       389,402.90 msec task-clock                       #   10.293 CPUs utilized
          444,959      context-switches                 #    1.143 K/sec
           45,761      cpu-migrations                   #  117.516 /sec
           31,451      page-faults                      #   80.767 /sec
  156,781,788,022      cpu_atom/instructions/           #    0.13  insn per cycle              (29.04%)
  268,876,355,825      cpu_core/instructions/           #    0.17  insn per cycle              (66.84%)
1,246,123,703,016      cpu_atom/cycles/                 #    3.200 GHz                         (29.02%)
1,573,508,255,155      cpu_core/cycles/                 #    4.041 GHz                         (66.84%)
   13,808,955,626      cpu_atom/branches/               #   35.462 M/sec                       (29.03%)
   23,141,851,979      cpu_core/branches/               #   59.429 M/sec                       (66.84%)
       55,007,380      cpu_atom/branch-misses/          #    0.40% of all branches             (29.01%)
       73,532,710      cpu_core/branch-misses/          #    0.32% of all branches             (66.84%)
            TopdownL1 (cpu_core)                 #     88.2 %  tma_backend_bound
                                                 #      0.5 %  tma_bad_speculation
                                                 #      3.5 %  tma_frontend_bound
                                                 #      7.8 %  tma_retiring             (66.84%)
                                                 #      2.6 %  tma_bad_speculation
                                                 #      5.5 %  tma_retiring             (29.02%)
                                                 #     88.5 %  tma_backend_bound
                                                 #      3.3 %  tma_frontend_bound       (29.01%)

     37.833572805 seconds time elapsed

    383.913216000 seconds user
      5.199233000 seconds sys
```

1. Resource Utilization:-

Parallel Efficiency: With 10.293 CPUs utilized, the system is highly parallelized, distributing the massive 389.40 seconds of task-clock time across roughly 10 processor cores simultaneously during the 37.83-second elapsed physical time. **Hybrid Execution:** The workload is actively split between the Performance-cores (running at a high 4.041 GHz) and Efficiency/Atom-cores (running at 3.200 GHz).

2. Instruction Efficiency:-

P-Core Performance: The Performance cores show a catastrophic drop in IPC (Instructions Per Cycle), hitting just 0.17. Only a dismal 7.8% of operations are successfully retiring. This indicates the fast cores are spending almost all their time completely stalled.

E-Core Lag: The Efficiency (Atom) cores perform even worse, with an IPC of just 0.13 and an extremely low retiring rate of 5.5%.

Branch Prediction: Unlike the DNA sequence alignment, both core types handle branch prediction exceptionally well. Branch misses are practically non-existent at 0.32% for P-cores and 0.40% for E-cores. The finite difference loops used in heat

simulations are highly regular, so the CPU almost always guesses the control flow correctly.

3. Primary Bottlenecks:-

Catastrophic Backend Bound (88.2% on P-Cores, 88.5% on Atom): This is the defining characteristic of this heat simulation. Both core types are completely choked in the backend. With nearly 90% of the execution pipeline stalled on both the fast and slow cores, the processing units are entirely starved for data. They are spending almost all their time waiting on memory access (RAM) to fetch the 2D grid neighbors for the stencil computation, proving that the application is severely limited by memory bandwidth rather than compute capability. Because the loops are so predictable, time wasted on bad speculation is negligible (0.5% on P-cores and 2.6% on E-cores). The CPU is rarely throwing away bad work; it is simply forced to wait for data to arrive.